Robust and Efficient Cartesian Mesh Generation for Component-Based Geometry

M.J. Aftosmis[*]
U.S. Air Force Wright Laboratory / NASA Ames
Moffett Field, CA 94035

M.J. Berger[†]
Courant Institute
New York, NY 10012

J.E. Melton[‡]
NASA Ames Research Center
Moffett Field, CA 94035

Abstract

This work documents a new method for rapid and robust Cartesian mesh generation for component-based geometry. The new algorithm adopts a novel strategy which first intersects the components to extract the wetted surface before proceeding with volume mesh generation in a second phase. The intersection scheme is based on a robust geometry engine that uses adaptive precision arithmetic and which automatically and consistently handles geometric degeneracies with an algorithmic tie-breaking routine. The intersection procedure has worse case computational complexity of $O(N \log N)$ and is demonstrated on test cases with up to 121 overlapping and intersecting components including a variety of geometric degeneracies.

The volume mesh generation takes the intersected surface triangulation as input and generates the mesh through cell division of an initially uniform coarse grid. In refining hexagonal cells to resolve the geometry, the new approach preserves the ability to directionally divide cells which are well-aligned with local geometry. The mesh generation scheme has linear asymptotic complexity with memory requirements that total approximately 14 words/cell. The mesh generation speed is approximately 10^6cells/minute on a 195Mhz RISC R10000 workstation.

I. Introduction

The past several years have seen a large resurgence of interest in adaptive Cartesian mesh algorithms for application to problems involving complex geometries. Refs. [1-12], among others, have proposed flow solvers and mesh generation schemes intended for use with arbitrary geometries. Since generating suitable Cartesian meshes is relatively quick, and the process can be fully automated, much of the on-going research focuses on quick extraction of CFD-ready geometry from the CAD databases to provide easy access to accurate solutions of Euler equations within the design cycle.

Viewing configurations on a component basis has several conceptual advantages over treatments which work with a single complete configuration. The most obvious of these is that components can be translated/rotated with respect to one another without requiring user intervention or a time consuming return to CAD in order to extract new intersection information and a new CFD-ready description of the wetted surface. Many approaches begin with a surface triangulation already constrained to the intersection curves of the components[9]. By starting upstream in the process, the component-based approach requires only that each piece of the geometry be described as a single closed entity. Thus, relative motion of parts may be pre-programmed or even computed as a result of a design analysis. The approach offers obvious advantages for automation through external, macroscopic control.

This flexibility comes at the expense of added complexity within the grid generation process. Since components may overlap, the possibility exists that a Cartesian cell-surface intersection detected during mesh generation may be entirely internal to the configuration, and thus all such intersections must be classified as "exposed" (retain) or "internal" (reject). Even if the vast majority of such intersections are actually part of the wetted surface, *all* intersections have the possibility of being internal, and therefore must be tested. An analysis of the mesh generation procedure documented in Refs.[1] and [13] revealed that up to 60% of the computation was dedicated to the resolution of this issue.

Although several approaches toward streamlining the process exist, the most attractive appears to be one which avoids the issue of intersection classification altogether. By first intersecting all components together, one can extract precisely the wetted surface so that all subsequent Cartesian cell intersections are guaranteed to be "exposed" and therefore retained. The remaining mesh generation problem may then be treated as if it were a single component problem.

While conceptually straightforward, efficient implementation of such an intersection algorithm is delicate. Each component is assumed to be described by a surface triangulation and the solution involves a sequence of problems in computational geometry. The

[*]. Aerospace Engineer, Senior Member AIAA
[†]. Prof. Dept. of Comp. Sci., Member AIAA
[‡]. Aerospace Engineer, Senior Member AIAA

algorithm requires intersecting a number of non-convex polyhedra with arbitrary genus. This makes convex polyhedra intersection algorithms inappropriate. Each intersected triangle must be broken up into smaller ones, which is a problem in constrained triangulation. Finally, the deletion of the interior triangles requires inside/outside determination and neighbor painting. Since intersecting triangles from different components must be considered to be in general position, the specters of robustness and finite precision mathematics must be considered as well. Section II presents a robust algorithm for computing these intersections and extracting the wetted surface on realistically complex examples. This algorithm is quite general and has numerous applications outside the specific field of CFD.

Section III presents the volume mesh generation algorithm with particular attention to the efficiency of data structures and the speed of intersection tests. The approach preserves the ability to directionally refine mesh cells. This feature comes in response to earlier work which concluded that isotropic cell division may lead to excessive numbers of Cartesian cells in three dimensions[13]. This work suggested that lower dimensional features may frequently be resolved by directional division of Cartesian cells.

II. Component Intersection

The problem of intersecting the various components of a given configuration and extracting the wetted surface may be viewed as a series of smaller problems not uncommon in computational geometry. This section briefly discusses some of the key aspects involved in the process. Focus centers on the topics of proximity searching, primitive geometric operations, exact arithmetic and algorithms for breaking geometric degeneracies.

A. Proximity Queries

Without special care, the intersection algorithm can result in implementations which have an asymptotic complexity of $O(N^2)$. The primary culprit here is the repetition of geometric searches to determine a list of candidate triangles on all components which may intersect with a given triangle on the polyhedron under consideration.

A number of data structures have been proposed to speed up this process, and one particularly suitable method is the Alternating Digital Tree (ADT) algorithm developed in Ref.[14]. Inserting the triangles into an ADT makes it possible to identify the list of candidate triangles in $O(\log N)$ operations. As a result, intersections need only be checked against candidate triangles from the list of spanning triangles returned from the tree. The basic algorithm outlined here follows from [14] and is implemented using the balanced tree approach detailed in Ref. [13].

The ADT is a hyperspace search technique which converts the problem of searching for finite sized objects in d dimensions to the simpler one of partitioning a space with $2d$ dimensions. Since searches are not conducted in physical space, objects which are physically close together are not necessarily close in the search space. This fact can hamper the tree's performance in some instances[15]. In an effort to improve lookup times, we therefore first apply a component bounding-box filter on the triangles before inserting them into the tree. Since they cannot possibly participate in an intersection, triangles which are not contained by the bounding box of a component other than their own are not inserted into the tree. This filtering not only reduces the tree size but also improves the probability of encountering an intersection candidate within the tree, since the structure is not crowded with irrelevant geometry.

B. Intersection of Generally Positioned Triangles in R^3

With the task of intersecting a particular triangle reduced to an intersection test between that triangle and those on the list of candidates provided by the ADT, the intersection problem is re-cast as a series of tri-tri intersection computations. Figure 1 shows a view of two intersecting triangles as a model for discussion. Each intersecting tri-tri pair will contribute one segment to the final polyhedra that will comprise the wetted surface of the configuration. The assumption of data in *general* (as opposed to *arbitrary)* position implies that the intersection is always non-degenerate. Triangles may not share vertices, and edges of tri-tri pairs do not intersect exactly. Thus, all intersections will be proper. This restriction will be lifted in later sections with the introduction of an automatic tie-breaking algorithm.

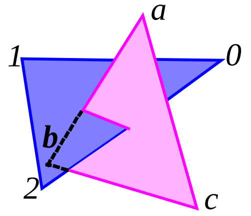

Figure 1: An intersecting pair of generally positioned triangles in three dimensions.

Several approaches exist to compute such intersections but a particularly attractive technique comes in the form of a Boolean test. This predicate can be performed robustly and quickly using only multiplication and addition, thus avoiding the inaccuracy and robustness pitfalls associated with division using fixed width representations of floating point numbers. It is useful to present a rather comprehensive treatment of this intersection primitive because subsequent sections on robustness will return to these relations.

For two triangles to properly intersect in three dimensional space, the following conditions must exist:

1. Two edges of one triangle must cross the plane of the other.
2. If condition (1) exists, there must be a total of two edges (of the six available) which pierce within the boundaries of the triangles.

One approach to checking these conditions is to directly compute the pierce points of the edges of one triangle in the plane of the other. Pierce locations from one triangle's edges may then be tested for containment within the boundary of the other triangle. This approach, while conceptually simple, is error prone when implemented using finite precision mathematics. In addition to demanding special effort to trap out zeros, the floating point division required by this approach may result in numbers not exactly representable by finite width words. This results in a loss of control over precision and may cause serious problems with robustness.

An alternative to this slope-pierce test is to consider a Boolean check based on computation of a triple product without division. A series of such logical checks have the attractive property that they permit one to establish the existence and connectivity of the segments without relying on the problematic computation of the pierce locations. The final step of computing the locations of these points may then be relegated to post-processing where they may be grouped together and, since the connectivity is already established, floating point errors will not have fatal consequences.

The Boolean primitive for the 3D intersection of an edge and a triangle is based on the concept of the signed volume of a tetrahedron in R^3. This signed volume is based on the well established relationship for the computation of the volume of a simplex, T, in d dimensions in determinate form (see for ex. [16]). The signed volume $V(T)$ of the simplex T with vertices $(v_0, v_1, v_2, \ldots, v_d)$ in d dimensions is:

$$d!\,V(T_{v_0 v_1 v_3 \ldots v_d}) = \det\begin{bmatrix} v_{0_0} & v_{0_1} & \cdots & v_{0_{d-1}} & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ v_{d_0} & v_{d_1} & \cdots & v_{d_{d-1}} & 1 \end{bmatrix} \qquad (1)$$

where v_{k_j} denotes the j^{th} coordinate of the k^{th} vertex with $j, k \in \{0, 1, 2, \ldots, d\}$. In 3 dimensions, eq. {2} gives six times the signed volume of the tetrahedron T_{abcd}.

$$6V(T_{abcd}) = \begin{vmatrix} a_0 & a_1 & a_2 & 1 \\ b_0 & b_1 & b_2 & 1 \\ c_0 & c_1 & c_2 & 1 \\ d_0 & d_1 & d_2 & 1 \end{vmatrix} = \begin{vmatrix} a_0 - d_0 & a_1 - d_1 & a_2 - d_2 \\ b_0 - d_0 & b_1 - d_1 & b_2 - d_2 \\ c_0 - d_0 & c_1 - d_1 & c_2 - d_2 \end{vmatrix} \qquad (2)$$

This volume serves as the fundamental building block of the geometry routines. It is positive when (a,b,c) forms a counterclockwise circuit when viewed from an observation point located on the side of the plane defined by (a,b,c) which is opposite from d. Positive and negative volumes define the two states of the Boolean test while zero indicates that the four vertices are exactly coplanar. If the vertices are indeed coplanar, then the situation constitutes a "tie" which will be resolved with a general tie-breaking algorithm presented shortly. In applying this logical test to edge ab and triangle (0,1,2) in fig. 1, ab crosses the plane if and only if (iff) the signed volumes T_{012a} and T_{012b} have opposite signs. Figure 2 presents a graphical look at the application of this test.

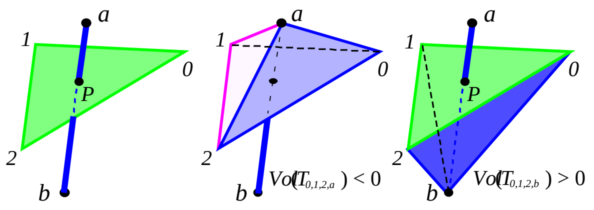

Figure 2: Boolean test to check if edge ab crosses the plane defined by triangle (0,1,2) through computation of two signed volumes.

With a and b established on opposite sides of the plane (0,1,2), all that remains is to determine if ab pierces within the boundary of the triangle. This will be the case only if the three tetrahedra formed by connecting the end points of ab with the three vertices of the triangle (0,1,2) (taken two at a time) all have the same sign, that is:

$$[V(T_{a12b}) < 0 \wedge V(T_{a01b}) < 0 \wedge V(T_{a20b}) < 0] \quad \text{or} \\ [V(T_{a12b}) > 0 \wedge V(T_{a01b}) > 0 \wedge V(T_{a20b}) > 0] \qquad (3)$$

Figure 3 illustrates this test for the case where the three volumes are all positive.

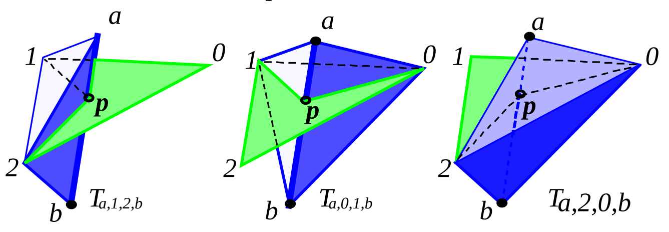

Figure 3: Boolean test for pierce of a line segment ab within the boundary of a triangle $(0,1,2)$.

After determining the existence of all the segments which result from intersections between tri-tri pairs and connecting a linked list of all such segments to the triangles that intersect to produce them, all that remains is to actually compute the locations of the pierce points. This is accomplished by using a parametric representation of each intersected triangle and the edge which pierces it. The technique is a straightforward three dimensional generalization of the 2D method presented in reference [16].

C. Retriangulation of Intersected Triangles

The final result of the intersection step is a list of segments linked to each intersected triangle. These segments divide the intersected triangles into polygonal regions which are either completely inside or outside of the body. In order to remove the portions of these triangles which are inside, we triangulate these polygonal regions within each intersected triangle and then reject the triangles which lie inside the body. Figure 4 shows a typical intersected triangle divided into two polygonal regions with the segments resulting from the intersection calculation highlighted. In the sketch, the two polygonal regions formed by the triangle's boundary and the segments from the intersection have been decomposed into sets of triangles. Since the segments may cut the triangle arbitrarily, a pre-disposition for creating triangles with arbitrarily small angles exists. In an effort to keep the triangulations as wel behaved as possible, we employ a two dimensional Delaunay algorithm within each original intersected triangle. Using the intersection segments as constraints, the algorithm runs within each intersected triangle producing new triangles which may be uniquely characterized as either inside or outside of the configuration.

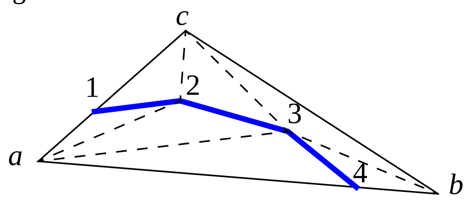

Figure 4: Decomposition of intersected triangle using a constrained Delaunay triangulation algorithm (constraining segments shown as heavy solid lines).

A variety of approaches to constructing the Delaunay triangulation of a planar graph exist (see surveys in Refs.[17],[18]). However, since each triangulation to be constructed starts with the three vertices of the original intersected triangle (vertices a,b,c in Fig. 4) the incremental algorithm of Green and Sibson[19] is appealing. Starting with the three vertices defining the original triangle, the pierce points are successively inserted into the evolving triangulation. After all the pierce points are inserted, the constraints are enforced using a simple recursive edge swapping routine.

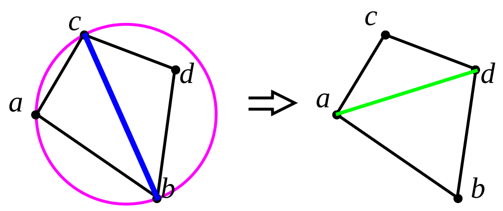

Figure 5: Incircle testing of point d for containment within the circumcircle of (a,b,c). Since d is contained, the diagonal of the quadrilateral $abcd$ is swapped $(cb \to ad)$.

The Green and Sibson algorithm is extensively documented in the literature. Its central operation is the local application of an "incircle predicate" which determines which edges need swapping to make the triangulation Delaunay after each successive point insertion. This predicate examines the four points of a quadrilateral formed by two triangles which share a common edge. In figure 5, if the point d falls within the circle defined by (a,b,c) then the diagonal of the quad must be swapped $(cb \to ad)$.

Relating this discussion to the signed volume calculation of eq.{2} starts by recognizing that if one projects the 2D coordinates (x,y) of each point in the incircle test onto a unit paraboloid $z = x^2 + y^2$ with the mapping:

$$(k_x, k_y) \to (k_x, k_y, k_x^2 + k_y^2) \quad \text{with} \quad k \in \{a, b, c, d\} \tag{4}$$

then the four points of the quadrilateral form a tetrahedron in 3D, and the incircle predicate may be viewed precisely as an evaluation of the volume of this tetrahedron. If $V(T_{a',b',c',d'}) > 0$ then point d lies within the circle defined by (a,b,c) and edge cb must be swapped to da for the triangulation to be Delaunay.

Figure 6 shows an example of this procedure applied within a single fuselage triangle which has been intersected by a wing leading edge. This example is interesting in that it demonstrates the need for robustness within the intersection and retriangulation algorithms. In this example, the wing leading edge has pierced a triangle of the fuselage. The intersection involves 52 constraining segments. Component data is considered "exact" in single precision, and the intersection points are computed using double precision. This example involved no tie-breaking, however, the succession of embedded enlargements in fig.6 make it clear that irregularity of the resulting triangulations demand robust implementation of the fundamental geometry routines.

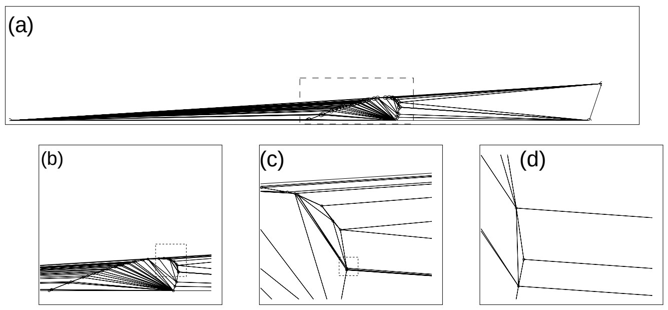

Figure 6: Retriangulation within a large fuselage triangle pierced by a wing leading edge component with significantly higher resolution. The 52 segments describing the intersection of the leading edge constrain the triangulation.

D. *Inside/Outside Determination*

The intersection and constrained triangulation routines of the two preceding sections have resulted in a set of triangles which may now be uniquely classified as either *internal* or *exposed* on the wetted surface of the configuration. The only step left is then to delete those triangles which are internal to the configuration. This is a specific application of the classic "point in polyhedron" problem for which we adopt a ray-casting approach. This method fits particularly well within the framework of proximity testing and primitive geometric computations described in sections II.A-B.

Simply stated, we determine if a point $p=(p_0, p_1, p_2)$ is within the i^{th} component of a configuration Ω_i if a ray, r, cast from p intersects the closed boundary Ω_i an odd number of times. The preceding two sections demonstrated that both the intersection and triangulation algorithms could be based upon Boolean operations checking the sign of the 3 x 3 determinant in eq.{2}, and the same is true for the ray casting step. Assuming that r is cast along a coordinate axis ($+x$ for example) it may be truncated just outside the $+x$ face of the bounding box for the entire configuration $\lceil \partial\Omega \rceil_x$ This ray may then be represented by a line segment from the test point (p, p_1, p_2) to ($\lceil \partial\Omega \rceil_x + \varepsilon$, p_1, p_2) and the problem reduces to a proximity query as in II.A and the segment-triangle intersection algorithm of II.B. The ADT returns the list of intersection candidates while the logical tests of Figs. 2 and 3 use eq.{2} to check for intersections. Counting the number of such intersections determines a triangle's status as inside or outside.

To avoid casting as many rays as there are triangles, a "painting algorithm" allows each tested triangle to pass on its status to the three triangles which share its edges. The algorithm then recurses upon the neighboring triangles until a constrained edge is encountered at which time it stops. In this way the entire configuration is "painted" using very few ray casts. The recursive algorithm is implemented using a stack to avoid the overhead associated with recursion.

Figures 7 and 8 present two brief examples. Figure 7 is a helicopter example problem containing 82 components with 320,000 triangles. The configuration includes external stores, and armaments. The complete intersection, retriangulation, and removal of interior geometry required ~200sec on workstation with a 195 Mhz MIPS R10000 processor. Figure 8 shows two close-ups of the inboard nacelle on a high-wing transport configuration. The frame on the left shows the final geometry after intersection, retriangulation, and trimming while the right frame shows a view inside by removing the outboard section of the wing with a cutting plane through the center of the nacelle. This configuration consisted of 86 components described by 214,000 triangles.

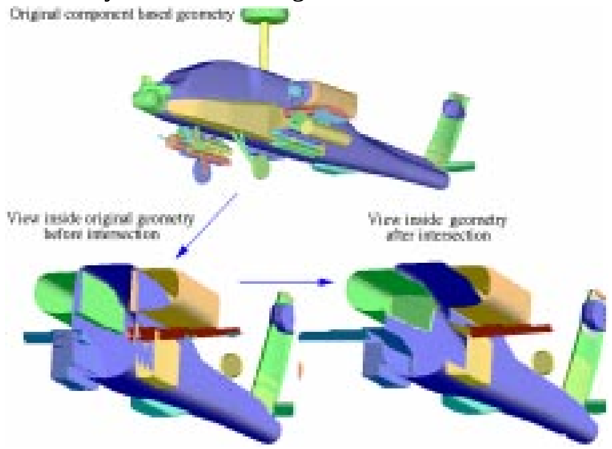
Original component based geometry
View inside original geometry before intersection
View inside geometry after intersection

Figure 7: Helicopter example containing 82 components ncluding external stores and armaments.

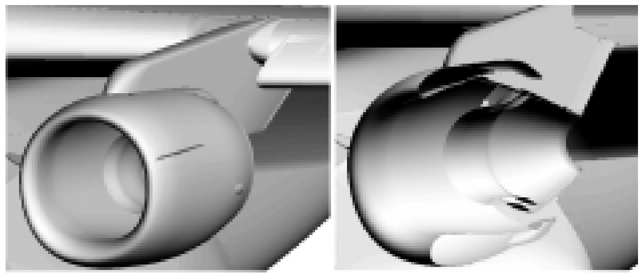

Figure 8: (left) Close-up of inboard nacelle of high-wing transport after intersection, retriangulation and removal of interior geometry, frame shows leading edge slat, wing section, pylon nacelle, nacelle strakes, and various other engine components. (right) view inside after removal of internal geometry. The configuration consisted of 86 components described using 214,000 triangles

E. *Geometric Degeneracies, Floating Point Arithmetic and Tie-Breaking*

With the intersection, retriangulation and ray casting algorithms all wholly dependent upon the determinant computation of eq.{2}, it is imperative to insure accurate evaluation of this volume. In fact, all operations which establish the connectivity of the final exterior surface triangulation involve computation of the sign of this determinant by design. As a result of this choice, the robustness of the overall procedure ultimately equates to a robust implementation of the signed volume calculation. Fortunately, evaluation of this determinant has long been the subject of study in computational geometry and computer science[20][21][22].

Computing the sign of eq.{2} constitutes a *topological primitive*, which is an operation that tests an input and results in one of a constant number of cases. Such primitives can only classify, and constructed objects (like the actual locations of the pierce points in II.B) cannot be determined without further processing. These primitives do, however, provide the intersections implicitly, and this information suffices to establish the connectivity of the segment list describing the intersection.

The signed volume computation for arbitrarily positioned geometry can return a result which is positive (+1), negative (–1) or zero (0), where ± 1 are non-degenerate cases and zero represents some geometric degeneracy. Implementation of this predicate, however, can be tricky, since it requires that we distinguish round-off error from an exact zero. Such considerations usually lead practitioners to implement the predicate with exact (arbitrary precision) arithmetic or with strictly integer math. Unfortunately, while much hardware development has gone into rapid floating point computation, few hardware architectures are optimized for either the arbitrary precision or integer math alternatives.

Floating Point Filtering and Exact Arithmetic

In an effort to perform as much of the computation as possible on the floating-point hardware, we first compute eq.{2} in floating point, and then make an *a-posteriori* estimate of the maximum possible value of the round-off error, $\varepsilon_{RE\,max}$. If this error is larger than the computed signed volume, then the case is considered indeterminate and we invoke the adaptive precision exact arithmetic procedure developed by Shewchuk[22]. If the case turns out to be identically zero, we then resolve the degeneracy with a general tie-breaking algorithm based on a virtual perturbation approach.

An error bound, $\varepsilon_{RE\,max}$, for floating point computation of the 3x3 determinant in eq.{2} was derived in Ref.[22]. The derivation accounts not only for the error in computing the determinant, but also for the error associated with floating point computation of the bound itself. This bound may be expressed as:

$$\begin{aligned} \varepsilon_{REmax} &= (7\varepsilon + 56\varepsilon^2) \otimes (\alpha_A \oplus \alpha_B \oplus \alpha_C) \qquad \text{with} \\ \alpha_A &= |a_2 \ominus d_2| \otimes (|(b_0 \ominus d_0) \otimes (c_1 \ominus d_1)| \oplus |(b_1 \ominus d_1) \otimes (c_0 \ominus d_0)|) \\ \alpha_B &= |b_2 \ominus b_2| \otimes (|(c_0 \ominus d_0) \otimes (a_1 \ominus d_1)| \oplus |(c_1 \ominus d_1) \otimes (a_0 \ominus d_0)|) \\ \alpha_C &= |c_2 \ominus d_2| \otimes (|(a_0 \ominus d_0) \otimes (b_1 \ominus d_1)| \oplus |(a_1 \ominus d_1) \otimes (b_0 \ominus d_0)|) \end{aligned} \tag{5}$$

where the circle (o) overstrike on +, -, and x indicates that the operations are evaluated using floating point operations on IEEE 754 compliant hardware. ε in eq.{5} is precisely $\varepsilon = 2^{-p}$ where p is the number of bits of the significand used by the machine. p may be evaluated by determining the largest exponent for which $1.0 \oplus 2^{-p} = 1.0$ when the sum and equality are tested with floating point. On most 32-bit platforms with exact rounding $p = 53$ for double precision and $p = 24$ for single.

In practice, only a very small fraction of the determinant evaluations fail to pass the test of eq.{5}. For the helicopter example problem shown earlier in Figure 7, the intersection required 1.37M evaluations of the determinant, and of these, only 68 (0.005%) failed to pass the floating point filter of eq.{5}.

Tie-Breaking and Degeneracy

With degenerate geometry identified by the exact arithmetic routines, we must now remove the restriction imposed by the initial assumption that all input geometric data lie in general position. The richness of possible degeneracies in three dimensions cannot be overstated, and without some systematic method of identifying and coping with them, handling of special cases can permeate, or even dominate the design of a geometric algorithm[23]. Rather than attempt to implement an *ad-hoc* tie-breaking algorithm based on intuition and programmer skill, we seek an algorithmic approach to this problem.

Simulation of Simplicity (SoS) is one of a category of general approaches to degenerate geometry known generically as “virtual perturbation algorithms”[24]. The basic premise is to assume that all input data undergoes a unique, ordered perturbation such that all ties are broken (*i.e.* data in special position is perturbed into general position). When a tie (an exact zero in eq.{2}) is encountered, we rely on the perturbations to break the tie. Since the perturbations are both unique and constant, any tie in the input geometry will always be resolved in a topologically consistent manner. Since the perturbations are virtual, no given geometric data is ever altered.

The perturbation $\varepsilon(i, j)$ at any point is a function of the point’s index, $i \in \{0, 1, \ldots, N-1\}$ and the coordinate direction, $j \in \{1, \ldots, d\}$. Ref. [24] advocates a perturbation of the form:

$$\varepsilon(i, j) = \varepsilon^{2^{i \cdot \delta - j}} \qquad \text{where} \qquad \begin{matrix} 0 \le i \le N-1 \\ 1 \le j \le d \\ \delta \ge d \end{matrix} \tag{6}$$

This choice indicates that the perturbation applied to i_j is greater than that on k_l iff $(i < k)$ or $(i = k) \wedge (j > l)$.

To illustrate, consider the two dimensional version of the simplex determinant in eq.{2}.

$$\det[M] = \det\begin{pmatrix} a_0 & a_1 & 1 \\ b_0 & b_1 & 1 \\ c_0 & c_1 & 1 \end{pmatrix} \tag{7}$$

If the points a,b,c are assumed to be indexed with $i = 0, 1, 2$ respectively, then taking $\delta = 2$ gives a perturbation matrix with:

$$\Lambda = \begin{pmatrix} \varepsilon^{2^{-1}} & \varepsilon^{2^{-2}} & 1 \\ \varepsilon^{2^{2-1}} & \varepsilon^{2^{2-2}} & 1 \\ \varepsilon^{2^{4-1}} & \varepsilon^{2^{4-2}} & 1 \end{pmatrix} \tag{8}$$

Taking the determinant of the perturbed matrix $M_\Lambda = M + \Lambda$ yields:

$$\begin{aligned} \det[M_\Lambda] = \det[M] &+ \varepsilon^{1/4}(-b_0 + c_0) \\ &+ \varepsilon^{1/2}(b_1 - c_1) + \varepsilon^{1}(a_0 - c_0) \\ &+ \varepsilon^{3/2}(1) + \varepsilon^{2}(-a_1 + c_1) \\ &+ \varepsilon^{9/4}(-1) + HOT \end{aligned} \tag{9}$$

Since the data, a,b,c span a finite region in 2-space, intuitively one can always choose a perturbation small enough such that increasing powers of ε always lead to terms with decreasing magnitude. Ref.[24] shows that this observation holds for a perturbation of the form of eq.{6}. If $\det[M]$ ever evaluates to an exact zero, the sign of the determinant will be determined by the sign of the next significant coefficient in the ε expansion. If the next term also yields an exact zero, we continue checking the signs of the coefficients until a non-zero term appears. In eq.{9} the coefficient on the fifth term ($\varepsilon^{3/2}$) is a constant (–1) and since *sign*(-1) is always negative, this term will never be degenerate.

The three dimensional variant of the simplex determinant (eq.{2}) has 15 non-zero coefficients before the first constant is encountered. Appendix I lists the hierarchy of terms in the 3-D expansion.

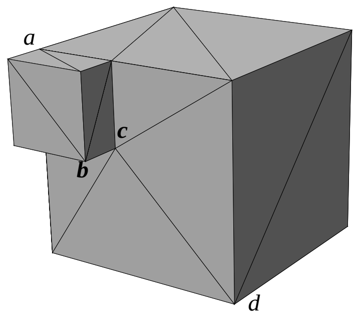

Figure 9: Two improperly intersecting right parallelepipeds with degeneracies resolved using virtual perturbations and exact arithmetic.

Figure 9 contains a deceptively simple looking application of the tie-breaking algorithm. The large and small cubes in the sketch abut against each other exactly. In addition to sharing the same geometry at location a, the cubes not only have three co-planar faces, but also have exact improper intersections where edge bc abutts against ad. The figure shows the result after computing the intersection, re-triangulating, and extracting the wetted surface. SoS resolved this case by imposing virtual perturbations such that the two polyhedra overlapped properly, consistently resolving not only the co-planar degeneracy, but also all improper *edge-edge* intersections. This geometry required 504 evaluations of eq.{2}, 186 of which evaluated to zero and required SoS for tie-breaking

III. Volume Mesh Generation

Generation of the Cartesian volume mesh begins with the final wetted surface triangulation resulting from the process in section II. This approach relieves the volume mesher of concerns about internal geometry and thus substantially reduces the complexity of the task.

A. Approach

The domain for a coordinate aligned Cartesian mesh may be defined by its minimum and maximum coordinates $\bar{x}_o$ and $\bar{x}_1$. Initially, this region is uniformly discretized with N_j divisions in each Cartesian dimension, $j = (0, 1, \ldots, d-1)$. By repeatedly dividing body intersecting cells and their neighbors, a final geometry-adapted mesh is obtained.

The mesh is considered to be an unstructured collection of Cartesian hexahedra. While many authors elect to traverse adaptively refined Cartesian meshes with an octtree data structure,[4][9][11] adopting an unstructured approach more readily preserves the ability to directionally refine the cells. This flexibility can be important since earlier work has suggested that permitting only isotropic refinement in three dimensions may lead to excessive numbers of cells for geometries with many length scales and high aspect ratio components[13].

Proximity Testing

Initially, the N_T surface triangles in $\{T\}$ are inserted into an ADT. This insures that returning the subset $T_{i.}$ of triangles actually intersected by the i^{th} Cartesian cell will have complexity proportional to $\log(N_T)$. When a cell is subdivided, a child cell inherits the triangle list of its parent. As the mesh subdivision continues, the triangle lists connected to a surface intersecting (“cut”) Cartesian cell will get shorter by approximately a factor of 4 with each successive subdivision. This observation implies that there is a machine dependent crossover at which it becomes faster to perform an exhaustive search over a parent cell’s triangle list, rather than perform an ADT lookup to get a list of intersection candidates for cell i. This crossover level is primarily determined by the number of elements in N_T and the processor’s instruction cache size. Conducting searches over the parent's triangle list implies that progressively smaller Cartesian cells may be intersected against T with ever decreasing computational complexity. For the large example problems in this paper, the crossover from ADT to exhaustive lookup usually occurs for cells with about $2000 < N_{T_i} < 5000$.

Geometric Refinement

All surface intersecting Cartesian cells in the domain are initially automatically refined a specified number of times $(R_{min})_j$. By default this level is set to be 4 divisions less than the maximum allowable number of divisions $(R_{max})_j$ in each direction. When a cut cell is tagged for division, the refinement is propagated several (usually 3-5) layers into the mesh using a "buffering" algorithm which operates by sweeps over the faces of the cells.

Further refinement is based upon a curvature detection strategy similar to that originally presented in Ref.[1]. This is a two-pass strategy which first detects angular variation of the surface normal, $\hat{n}$, within each cut cell and then examines the average surface normal behavior between two adjacent cut cells.

Taking k as a running index to sweep over the set of triangles, T_i, V_j is the j^{th} component of the vector subtraction between the maximum and minimum normal vectors in each Cartesian direction.

$$V_j = \max(n_{k_j}) - \min(n_{k_j}) \qquad \forall(k \in T_i) \tag{10}$$

The min(-) and max(-) are performed over all elements of T_i. The angular variation within cell i is then simply the direction cosines of $\bar{V}$

$$\cos(\theta_{i_j}) = \frac{\max(n_{k_j}) - \min(n_{k_j})}{|\bar{V}|} \tag{11}$$

Similarly, $(\phi_j)_{r,s}$ measures the j^{th} component of the angular variation between any two adjacent cut cells r and s. With $\hat{n}_i$ denoting the unweighted unit normal vector within any cut cell i, the components of $\bar{\phi}_{r,s}$ are:

$$\cos(\phi_j)_{r,s} = \frac{|n_{j_r} - n_{j_s}|}{|\hat{n}_r - \hat{n}_s|} \tag{12}$$

If θ_j or ϕ_j in any cell exceeds an angle threshold (usually set to $25°$) the offending cell is tagged for subdivision in direction j.

B. Data Structures

Since we have adopted an unstructured approach and intend to construct meshes with 10^6 or 10^7 cells, its imperative that the data structures be as compact as possible. The system described in this section provides all cell geometry and cell-to-vertex pointers in 96 bits.

Figure [10] shows a model Cartesian mesh covering the region $[\bar{x}_0, \bar{x}_1]$. Every cell in such a mesh can be uniquely located by the integer coordinates (i_0, i_1, i_2) which correspond to the vertex closest to $\bar{x}_o$. If we allocate m bits of memory to store each integer i_j, the upper bound on the permissible total number of vertices in each coordinate direction is 2^m.

On a mesh with N_j prescribed nodes, performing R_j cell refinements in each direction will produce a mesh with a maximum integer coordinate of $2^{R_j}(N_j - 1) + 1$ which must be resolvable in m bits.

$$2^{R_j}(N_j - 1) + 1 \le 2^m \tag{13}$$

Thus, the maximum number of cell subdivisions that may be performed in each direction is:

$$(R_{max})_j = \left\lfloor \log_2(2^m - 1) - \log_2(N_j - 1) \right\rfloor \tag{14}$$

where the floor indicates rounding down to the next lower integer. Substituting back into eq.{13} gives the total number of vertices which we can address in each coordinate direction.

$$M_j = 2^{R_{max_j}}(N_j - 1) + 1 \tag{15}$$

Thus, the floor in eq.{14} insures that M_j can never exceed 2^m.

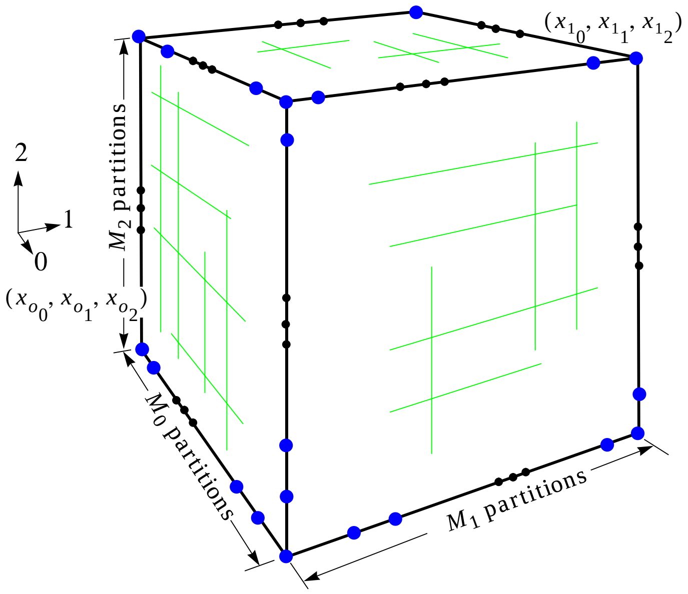

Figure 10: Cartesian mesh with M_j divisions in each direction discretizing the region from x_o to x_1.

Currently, we permit up to $m = 21$ bits per direction which gives about 2.1×10^6 addressible locations in each coordinate, while still permitting all three indices to be packed into a single 64-bit integer.

Figure [11] gives an example of the vertex numbering within an individual cell. This system has been adopted from the study of crystalline structures specialized for cubic lattices*. Within this analogy, the cell vertices are numbered with a boolean index of 0 (low) or 1 (high) in each direction. The crystal direction of each vertex is shown in square brackets. Reinterpreting this 3-bit pattern as an integer yields a unique numbering scheme (from 0-7) for each vertex on the cell.

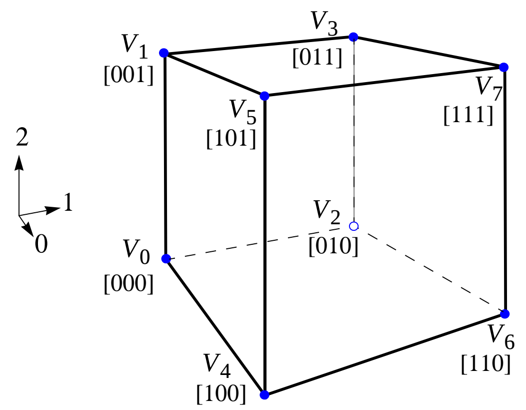

Figure 11: Vertex numbering with in a cell, numbers in square brackets [-] are the crystal directions of each vertex.

For any cell i, $\overline{V}_0$ is the integer position vector $(V_{0_0}, V_{0_1}, V_{0_2})$. If we also know the number of times that i has been divided in each direction, R_j, we can express its other 7 vertices directly.

$$\begin{aligned}
\overline{V}_1 &= \overline{V}_0 + (0, 0, 2^{Rmax_2 - R_2}) \\
\overline{V}_2 &= \overline{V}_0 + (0, 2^{Rmax_1 - R_1}, 0) \\
\overline{V}_3 &= \overline{V}_0 + (0, 2^{Rmax_1 - R_1}, 2^{Rmax_2 - R_2}) \\
\overline{V}_4 &= \overline{V}_0 + (2^{Rmax_0 - R_0}, 0, 0) \\
\overline{V}_5 &= \overline{V}_0 + (2^{Rmax_0 - R_0}, 0, 2^{Rmax_2 - R_2}) \\
\overline{V}_6 &= \overline{V}_0 + (2^{Rmax_0 - R_0}, 2^{Rmax_1 - R_1}, 0) \\
\overline{V}_7 &= \overline{V}_0 + (2^{Rmax_0 - R_0}, 2^{Rmax_1 - R_1}, 2^{Rmax_2 - R_2})
\end{aligned} \tag{16}$$

Since the powers of two in this expression are simply a left shift of the bitwise representation of the integer subtraction $Rmax_j - R_j$, vertices V_1-V_7 can be computed from V_0 and R_j at very low cost. In addition, the total number of refinements in each direction will be a (relatively) small integer, thus its possible to pack all three components of $\overline{R}$ into a single 32-bit word.

C. Cut-Cell Intersection

A central algorithm of any Cartesian mesh generation strategy involves testing the surface for intersection with the Cartesian cells. While the general edge-triangle intersection algorithm in section II.B would provide one method of testing for such intersections, a more attractive alternative comes from the literature on computer graphics[26][27].

This algorithm is highly specialized for use with coordinate aligned regions, and while it could be extended to non-Cartesian cells, or even other cell types, its speed and simplicity would be compromised. Since rapid cut-cell intersection is an important part of Cartesian mesh generation, we present a few central operations of this algorithm in detail.

Figure 12 shows a two dimensional Cartesian cell c which covers the region $[\bar{c}, \bar{d}]$. The points $(p, q,...,v)$ are assumed to be vertices of c's candidate triangle list T_c. Each vertex is assigned an "outcode" associated with its location with respect to cell c. This boolean code has 2 bits for each coordinate direction. Since the region is coordinate aligned, a single inequality must be evaluated to set each bit in the outcode of the vertices.

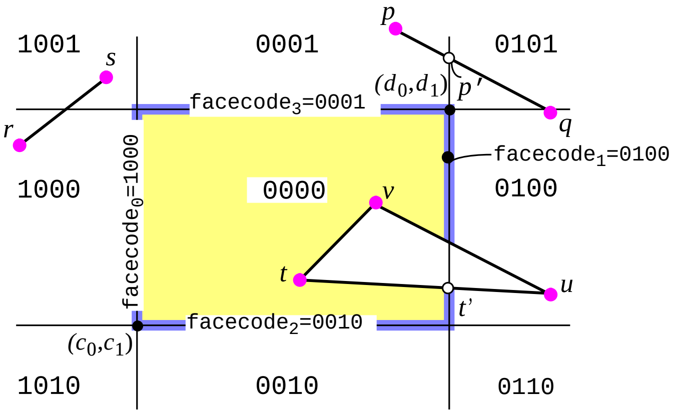

Figure 12: outcode and facecode setup of a coordinate aligned region $[\bar{c}, \bar{d}]$ in two dimensions.

Using the operators & and | to denote bitwise applications of the "and" and "or" boolean primatives, candidate edges (like rs) can be trivially rejected if

$$\texttt{outcode}_r \,\&\, \texttt{outcode}_s \neq 0$$

Similarly, since $(\texttt{outcode}_t \mid \texttt{outcode}_v) = 0$, the segment must be completely contained.

If all the edges of a triangle, like tuv, cannot be trivially rejected, then there is a possibility that it intersects the `0000` region. Such a polygon can be tested against the face-planes of the region by constructing a logical bounding box (using a bitwise "or") and testing against each `facecode` of the region. In Fig. 12 testing

$$\texttt{facecode}_j \,\&\, (\texttt{outcode}_t \mid \texttt{outcode}_u \mid \texttt{outcode}_v) \quad \forall j \in (0, 1, 2, ..., 2d\text{-}1) \tag{17}$$

results in a non-zero only for the `0100` face.

When an intersection point, such as p' or t', is computed, it can be classified and tested for containment on the boundary of $[\bar{c}, \bar{d}]$ by examination of its outcode. However, since these points lie degenerately on the `01XX` boundary, the contents of this bit may not be trustworthy. For this reason, we mask out the ques-

*. Such systems are quite general and can be used to describe cubic, orthorhombic, tetrahedral, or hexagonal cells. See [25].

tionable bit before examining the contents of these outcodes. Applying “not” in a bitwise manner yields:

$(\texttt{outcode}_{p'} \,\&\, (\neg\texttt{facecode}_1)) = 0$ while
$(\texttt{outcode}_{t'} \,\&\, (\neg\texttt{facecode}_1)) \neq 0$

which indicates that *t'* is on the face, while *p'* is not.

There are clearly many alternative approaches for implementing the types of simple queries that this section describes. However, an efficient implementation of these operations is central to the success of a Cartesian mesh code. The bitwise operations and comparisons detailed in the proceeding paragraphs generally execute in a single machine instruction making this a particularly attractive approach.

IV. Results

The intersection algorithm described in §II and the mesh generator of §III have been exercised on a variety of example problems. The presentation of numerical results includes several example meshes and an examination of the asymptotic performance of the algorithm. All computations were performed on a MIPS R10000 workstation with a 195Mhz CPU.

A. Example Meshes

High Speed Civil Transport

Figure 13 depicts two views of a 4.72M cell mesh constructed around a proposed supersonic transport design. This geometry consists of 8 polyhedra, two of which have non-zero genus. These components include the fuselage, wing, engine pylons and nacelles. The original component triangulation was comprised of 81460 triangles before intersection and 77175 after the intersection algorithm re-triangulated the intersections and extracted the wetted surface. Of the 1.2M calls placed to the determinant computation (eq.{2}), 870 were degenerate and required tie-breaking. The intersection required 15 seconds of workstation time.

The mesh shown contains 11 levels of cells where all divisions were isotropic. Mesh generation required 4 minutes and 20 seconds. The maximum memory required was 252Mb.

Helicopter Configuration

Figure 14 contains two views of a mesh produced for a non-symmetric helicopter configuration similar to that shown earlier in fig.7. This example began with 202000 triangles describing 82 components. After intersection and re-triangulation, 116000 triangles remained on the exterior. The final mesh contained 5.81M Cartesian cells with about 587000 actually intersecting the geometry. The mesh shown was refined 10 times to produce cells at 11 levels of refinement in the final mesh. Since the fuselage and wing components span the bounding boxes of most other components, virtually all triangles were loaded into the ADT resulting in a rather sluggish intersection computation which required just over 3 minutes of CPU time. The mesh was computed in approximately 5 minutes and 20 seconds.

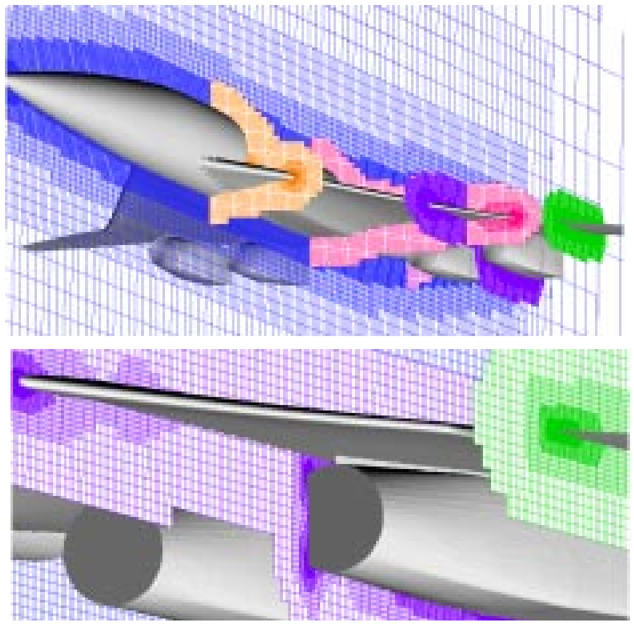
Figure 13: Upper: Cutting planes through 4.72M cell Cartesian mesh for a proposed HSCT geometry. Lower: Close-up of mesh near outboard nacelle. .

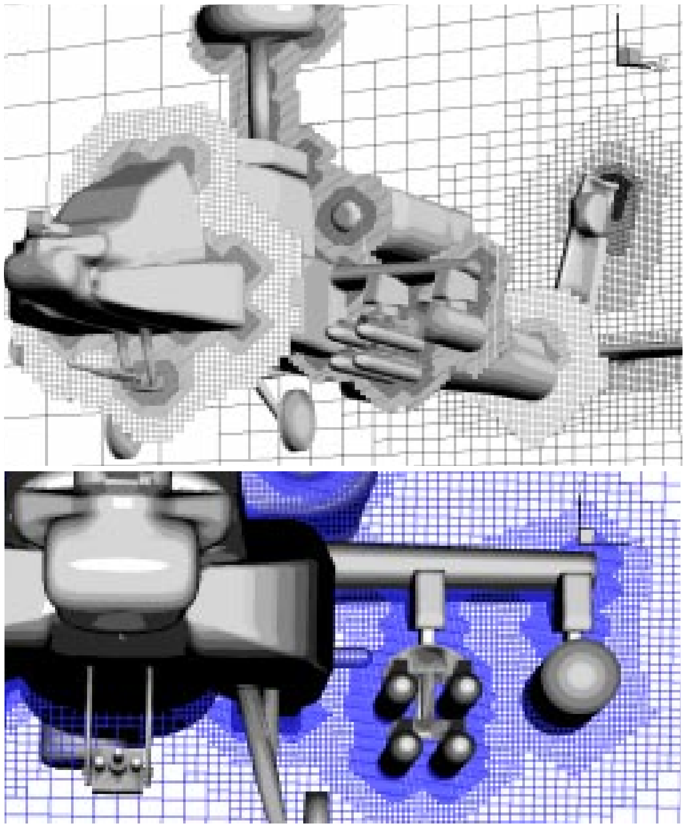
Figure 14: Upper: Cartesian mesh for attack helicopter configuration with 5.81M cells in the final mesh. Lower: close-up of mesh through left wing and stores.

Multiple Aircraft

The final mesh adds three twin tailed fighter geometries to the helicopter example. The helicopter is offset from the axis of the lead fighter to emphasize the asymmetry of the mesh. Each fighter has flow-through inlets and is described by 13 component triangulations. The entire configuration contained 121 components described with 807000 triangles before intersection and 683000 after. A total of 5916 determinant evaluations were degenerate and invoked the SoS routines.

Figure 15 presents an overview of the mesh. The upper frame shows portions of 3 cutting planes through the geometry. The lower frame in this figure shows one cutting plane at the tail of the rear two aircraft, and another just under the helicopter geometry. The final mesh includes 5.61M cells, and required a maximum of 365Mb to compute. Mesh generation time was approximately 6 minutes and 30 seconds.

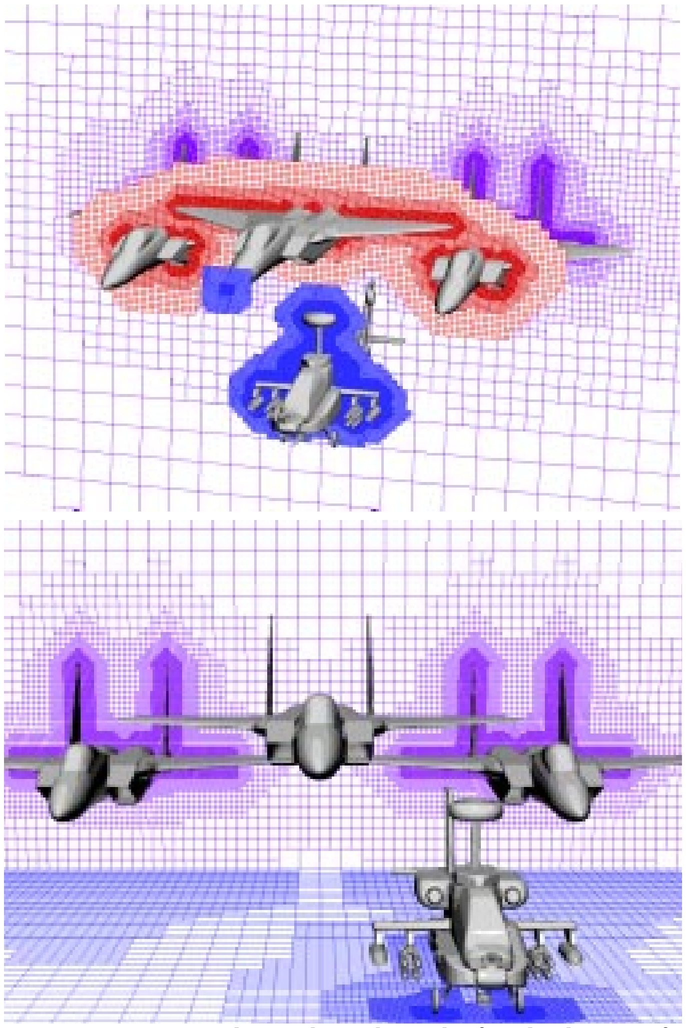
Figure 15: Cutting planes through mesh of multiple aircraft configuration with 5.61M cells and 683000 triangles in the triangulation of the wetted surface.

B. Asymptotic Performance

The number of triangles in the surface and the percentage of the mesh cells which are cut by this triangulation are two of the primary factors which effect mesh generation speed. The examples in the previous section have been chosen to demonstrate mesh generation speed for realistically complex geometries.

In order to assess the asymptotic behavior of the algorithm, the mesh generator was run on a teardrop geometry described by 7520 triangles. To prevent variation in the percentage of cut cells which are divided at successive refinements, the angle thresholds triggering mesh refinement were set to zero. This choice forced all cut cells to be tagged for refinement at every level.

A series of 11 meshes were produced for this investigation with between $7.5x10^3$ and $1.7x10^6$ cells in the final grids. The initial meshes used consisted of 6x6x6, 5x5x6, and 5x5x5 cells and were subjected to 3-9 levels of refinement.

Figure 16 contains a scatter plot of cell number vs. CPU time including file reading/writing. The solid line fitted to the data is the result of a linear regression. The line has a slope of $4.01x10^{-5}$ seconds/cell and a correlation coefficient of 0.9997. This equates to 24950 cells per second or $1.50x10^6$ cells per minute for this example. This strong correlation to a straight line indicates that the mesh generator produces cells with linear asymptotic complexity. This result is optimal for any method that operates cell-by-cell.

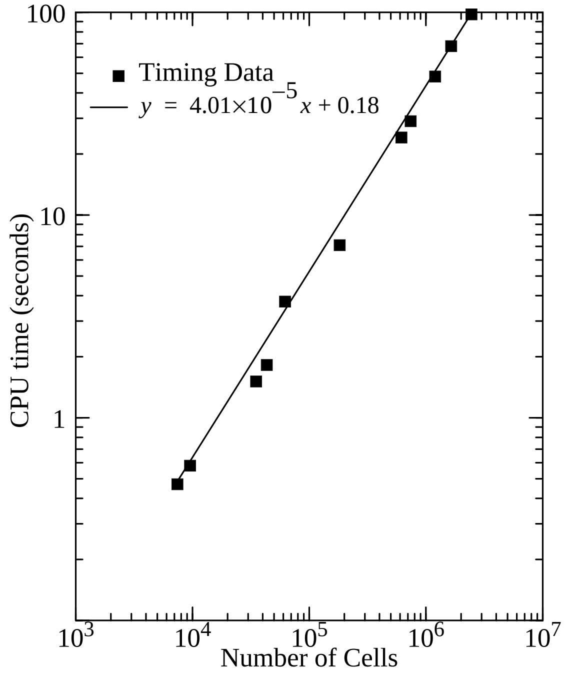

Figure 16: Scatter plot of mesh size vs. computation time. 195Mhz MISC R10000 CPU.

V. Conclusions and Future Work

We have developed a new Cartesian mesh generation algorithm for efficiently and robustly producing geometry adapted Cartesian meshes for complex configurations. The method uses a component-based approach for surface geometry and pre-processes the intersection between these components so that only the wetted surface is passed to the mesh generator.

The surface intersection algorithm uses exact arithmetic and adopts an algorithmic approach for handling degenerate geometry. The robustness of this approach was demonstrated on examples with nearly 6000 degenerate geometry evaluations.

The mesh generation algorithm was exercised on a variety of cases with complex geometries, involving up to 121 components described by 807000 triangles. The mesh generation operates on the order of 10^6 cells/minute on moderately powered workstations. Its memory usage is approximately 14 words/cell so that typically only 54Mb is required to generate a mesh of 1M cells. The example meshes contained up to 5.8M cells. An evaluation of the asymptotic performance of this algorithm indicated that cells are generated with linear computational complexity.

One aspect which has not been completely addressed is the degree to which anisotropic cell division can be used to improve the efficiency of adaptive Cartesian simulations on realistic geometries. Since the current method has the ability to refine cells directionally, this topic will be addressed in future work.

VI. Acknowledgments

M. Berger was supported by AFOSR grant 94-1-0132, and DOE grants DE-FG02-88ER25053 and DEFG02-92ER25139 and by RIACS at NASA Ames. The authors gratefully acknowledge the use of Jonathan Shewchuk's adaptive precision floating point software. In addition we acknowledge E.E-Chien and J. O'Rourke for helpful conversations, e-mail and other contributions.

VII. References

[1] Melton, J.E., *Automated Three-Dimensional Cartesian Grid Generation and Euler Flow Solutions for Arbitrary Geometries*. Ph.D. Thesis, U. of Ca, Dept. of Mech. Eng., Davis CA, Apr. 1996.

[2] Gaffney, R., Hassan, H., and Salas, M., "Euler Calculations for Wings Using Cartesian Grids." AIAA Paper 87-0356. Jan. 1987.

[3] Berger, M.J., and LeVeque, R.J., "An Adaptive Cartesian mesh Algorithm for the Euler Equations in Arbitrary Geometries." AIAA Paper 89-1930. Jun. 1989.

[4] Coirier, W.J., and Powell, K.G., "An Accuracy Assessment of Cartesian-Mesh Approaches for the Euler Equations." AIAA Paper 93-3335-CP. Jun. 1993.

[5] Quirk, J.J., "An Alternative to Unstructured Grids for Computing Gas Dynamic Flows around Arbitrarily Complex Two-Dimensional Bodies." Computers Fluids, Vol.23, No.1, pp. 125-142, 1994.

[6] Gooch, C.F., *Solution of the Navier-Stokes Equations on Locally-Refined Cartesian Meshes using State-Vector Splitting*. Ph.D. Dissertation, Department of Aeronautics and Astronautics, Stanford University, Palo Alto, CA., Dec. 1993.

[7] Melton, J.E., Enomoto, F.Y., and Berger, M.J., "3D Automatic Cartesian Grid Generation for Euler Flows." AIAA Paper 93-3386-CP. Jun. 1993.

[8] Melton, J.E., Berger, M.J., Aftosmis, M.J., and Wong, M.D., "3D Applications of a Cartesian Grid Euler Method." AIAA Paper 95-0853. Jan. 1995.

[9] Karman, S.L. Jr., "SPLITFLOW: A 3D Unstructured Cartesian/Prismatic Grid CFD Code for Complex Geometries." AIAA Paper 95-0343. Jan. 1995.

[10] Melton, J., Pandya, S., and Steger, J.L., "3D Euler Flow Solutions Using Unstructured Cartesian and Prismatic Grids." AIAA Paper 93-0331. Jan. 1993.

[11] Coirier, W.J., *An Adaptively Refined Cartesian Cell Method for the Euler and Navier-Stokes Equations*. Ph.D. Dissertation, Department of Aerospace Engineering, Univ. of Michigan, Ann Arbor MI, Sep. 1994.

[12] Pember, R.B., Bell, J.B., Colella, P., Crutchfield, W.Y., Welcome, M.L., "Adaptive Cartesian Grid Methods for Representing Geometry in Inviscid Compressible Flow." AIAA Paper 93-3385-CP. Jun.1993.

[13] Aftosmis, M.J., Melton, J.E., and Berger, M.J., "Adaptation and Surface Modeling for Cartesian Mesh Methods." AIAA Paper 95-1725-CP, Jun 1995.

[14] Bonet, J., and Peraire, J., "An Alternating Digital Tree (ADT) Algorithm for Geometric Searching and Intersection Problems." *Int. J. Num. Meth. Engng*, **31**:1-17, 1991.

[15] Samet, H., *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[16] O'Rourke, J., *Computational Geometry in C*. Cambridge Univ. Press, 1994.

[17] Mavriplis, D.J., "Unstructured Mesh Generation and Adaptivity." 26th Computational Fluid Dynamics Lecture Series, von Karman Institute, Mar 1995.

[18] Barth, T.J., "Aspects of Unstructured Grids and Finite-Volume Solvers for the Euler and Navier-Stokes Equations." 25th Computational Fluid Dynamics Lecture Series, von Karman Institute, Mar 1994.

[19] Green, P.J., and Sibson, R., "Computing the Dirichlet Tessalation in the Plane." *The Computer Journal*, **2**(21):168-173, 1977.

[20] Chvátal, V., *Linear Programming*. Freeman, San Francisco, Ca., 1983.

[21] Knuth, D.E., *Axioms and Hulls*. Lecture Notes in Comp. Sci. #606., Springer-Verlag, Heidelberg, 1992.

[22] Shewchuk, J.R., "Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates." *CMU-CS-96-140*, School of Computer Science, Carnegie Mellon Univ., 1996. currently also available at: *http://www.cs.cmu.edu/afs/cs/project/quake/public/papers/robust-predicates.ps*

[23] Chazelle, B., *et al.*, "Application Challenges to Computational Geometry: CG Impact Task Force Report." *TR-521-96*. Princeton Univ., Apr. 1996.

[24] Edelsbrunner H., and Mücke E.P., "Simulation of Simplicity: A Technique to Cope with Degenerate Cases in

Geometric Algorithms.” ACM Transactions on Graphics, **9**(1):66-104, Jan. 1990.

[25] Van Vlack, L.H., *Elements of Material Science and Engineering*. Addison-Wesley Inc. 1980.

[26] Cohen, E., “Some Mathematical Tools for a Modeler’s Workbench,” IEEE Comp. Graph. and App., **3**(7), Oct. 1983.

[27] Voorhies, D., *Graphics Gems II: Triangle-Cube Intersections*. Academic Press, Inc. 1992.

Appendix

Performing the ε expansion from section II.E on the 4×4 determinant in eq.{2} results in 17 non-constant coefficients before the first constant is encountered in a power of ε. Of these, 15 are unique.

To illustrate we compute the determinant of the perturbed three dimensional simplex matrix:

$$\det[M + \Lambda] = \det\left(\begin{bmatrix} a_0 & a_1 & a_2 & 1 \\ b_0 & b_1 & b_2 & 1 \\ c_0 & c_1 & c_2 & 1 \\ d_0 & d_1 & d_2 & 1 \end{bmatrix} + \begin{bmatrix} \varepsilon^{1/2} & \varepsilon^{1/4} & \varepsilon^{1/8} & 1 \\ \varepsilon^{4} & \varepsilon^{2} & \varepsilon^{1} & 1 \\ \varepsilon^{32} & \varepsilon^{16} & \varepsilon^{8} & 1 \\ \varepsilon^{256} & \varepsilon^{128} & \varepsilon^{64} & 1 \end{bmatrix}\right)$$

Table A.I lists the terms in this expansion.

Table A.1:

$\varepsilon^?$	coefficient
ε^0	$\det\begin{pmatrix} a_0 & a_1 & a_2 & 1 \\ b_0 & b_1 & b_2 & 1 \\ c_0 & c_1 & c_2 & 1 \\ d_0 & d_1 & d_2 & 1 \end{pmatrix}$
$\varepsilon^{1/8}$	$\det\begin{pmatrix} b_0 & b_1 & 1 \\ c_0 & c_1 & 1 \\ d_0 & d_1 & 1 \end{pmatrix}$
$\varepsilon^{1/4}$	$(-1)\det\begin{pmatrix} b_0 & b_2 & 1 \\ c_0 & c_2 & 1 \\ d_0 & d_2 & 1 \end{pmatrix}$
$\varepsilon^{1/2}$	$\det\begin{pmatrix} b_1 & b_2 & 1 \\ c_1 & c_2 & 1 \\ d_1 & d_2 & 1 \end{pmatrix}$
ε^{1}	$(-1)\det\begin{pmatrix} a_0 & a_1 & 1 \\ c_0 & c_1 & 1 \\ d_0 & d_1 & 1 \end{pmatrix}$
$\varepsilon^{5/4}$	$\det\begin{pmatrix} c_0 & 1 \\ d_0 & 1 \end{pmatrix}$
$\varepsilon^{3/2}$	$(-1)\det\begin{pmatrix} c_1 & 1 \\ d_1 & 1 \end{pmatrix}$
ε^{2}	$\det\begin{pmatrix} a_0 & a_2 & 1 \\ c_0 & c_2 & 1 \\ d_0 & d_2 & 1 \end{pmatrix}$
$\varepsilon^{5/2}$	$\det\begin{pmatrix} c_2 & 1 \\ d_2 & 1 \end{pmatrix}$
ε^{4}	$(-1)\det\begin{pmatrix} a_1 & a_2 & 1 \\ c_1 & c_2 & 1 \\ d_1 & d_2 & 1 \end{pmatrix}$
ε^{8}	$\det\begin{pmatrix} a_0 & a_1 & 1 \\ b_0 & b_1 & 1 \\ d_0 & d_1 & 1 \end{pmatrix}$
$\varepsilon^{33/4}$	$(-1)\det\begin{pmatrix} b_0 & 1 \\ d_0 & 1 \end{pmatrix}$
$\varepsilon^{17/2}$	$\det\begin{pmatrix} b_1 & 1 \\ d_1 & 1 \end{pmatrix}$
ε^{10}	$\det\begin{pmatrix} a_0 & 1 \\ d_0 & 1 \end{pmatrix}$
$\varepsilon^{21/2}$	(+1)